Protecting Data from Malicious Software

Matthew Schmid & Frank Hill
Cigital, Inc.
{mschmid, fhill}@cigital.com

Anup K. Ghosh
DARPA
aghosh@darpa.mil

Abstract

Corruption or disclosure of sensitive user documents can be among the most lasting and costly effects of malicious software attacks. Many malicious programs specifically target files that are likely to contain important user data. Researchers have approached this problem by developing techniques for restricting access to resources on an application-by-application basis. These so-called "sandbox environments," though effective, are cumbersome and difficult to use. In this paper, we present a prototype Windows NT/2000 tool that addresses malicious software threats to user data by extending the existing set of file-access permissions. Management and configuration options make the tool unobtrusive and easy to use. We have conducted preliminary experiments to assess the usability of the tool and to evaluate the effects of improvements we have made. Our work has produced an intuitive data-centric method of protecting valuable documents that provides an additional layer of defense beyond existing antivirus solutions.

1. Introduction

Despite the efforts of the antivirus community, malicious software continues to be a major threat to businesses and to individuals. Malicious software commonly appears innocuous and carries on its true purpose unbeknownst to its victim. A particularly dangerous form of malicious software (malware) is that which remains undetected while it continues to perform malicious actions. Viruses may attach themselves to ordinary programs without any noticeable effect on the original software. Trojans may stealthily steal valuable information and transmit it anywhere on the Internet. Malicious mobile code may surreptitiously destroy or steal files while a person browses the World Wide Web.

For the typical user, the greatest threat that malware poses is its ability to steal, modify, or destroy important data. The costs associated with damage to the operating system or other software is negligible when compared with the value of the information that a person has labored to produce. After all, the computer and its software are simply tools being used to aid in the creation of this data in the first place. While corporations may be legitimately concerned with side effects such as system availability and the resources required to eradicate an out-of-control virus, users are ultimately concerned with the integrity and confidentiality of the data contained in their files.

The Windows NT / 2000 security model provides users with the ability to protect sensitive documents from access by other users. The type of access control provided by Windows is known as discretionary access control (DAC) because the owner of a file is given discretion in determining the access permissions for that file. This is in contrast with mandatory access control (MAC) where file permissions are much more tightly controlled [16]. Figure 1 shows the Windows NT 4.0 dialog box that enables a user to select file permissions for a document.

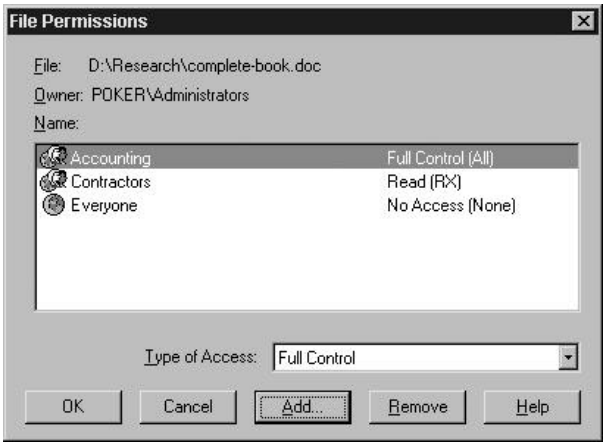

Figure 1. Discretionary access control on Windows NT

While this security model protects sensitive data from nosy or ill-intentioned colleagues, it does nothing to defend against most malicious software. The reason for this disparity is that malicious software is actually executed *by the user himself.* The consequence of a DAC

system is that once a person runs a program, that program has the ability to change the access permissions on any of that user's files. Malicious code therefore has the ability to do anything that the user could do, including the reading of, writing to, or destruction of any documents that belong to the compromised user.

In this paper we introduce the FileMonster; a prototype tool for extending Windows discretionary access control to better protect important files from damage or snooping by all forms of malicious software. The FileMonster shares some characteristics with prior research into application sandboxing (see the *Related Work* section), yet we believe that its unique approach to this problem solves many of the usability issues that have plagued earlier systems. This prototype provides an additional level of protection from malicious software with a minimal impact on the normal work environment.

2. A data-centric protection mechanism

Existing access control mechanisms do little to protect your files from malicious software. To make matters worse, the Windows operating system makes it difficult or impossible to observe how an application is using the file system without the help of a third-party monitoring tool (see *www.sysinternals.com* for several useful system monitoring tools). The prototype described here works by increasing the visibility of access to important documents on your computer. Essentially we provide two new file system permissions: *confirm on read* and *confirm on write*. These permissions indicate that when a program performs either a read or write operation on a protected file, the user must provide a confirmation before this operation can proceed. This eliminates the possibility that a malicious program can read or alter a protected file without first getting permission from the user. The *confirm on read* permission should be used on files whose contents are considered confidential. This will require a confirmation from a user if any program attempts to read data from the file. The *confirm on write* permission should be used on files whose valuable contents must not be damaged. This permission requires a confirmation when a program tries to write to or delete the file. These permissiond may also be combined (*confirm on access*). We have developed a prototype, named FileMonster, which enforces these permissions on the Windows NT/2000 operating system.

In addition to supporting permissions on a per file basis, the FileMonster also allows users to set permissions based on file types (extensions). This provides broad protection to a group of files. For example, a user might decide that all of his Microsoft Excel files (identified by the *XLS* file extension) should be protected with *confirm on read* permission. The FileMonster would then require confirmation whenever a program accessed a file with the extension of *XLS*. Individual file permissions always override group permissions, allowing someone to tailor file permissions to fit current needs.

We refer to our system as being *data-centric* to distinguish it from other research efforts that have investigated techniques of protecting resources against malicious software that have been largely *application-centric*. An application-centric approach to malicious software prevention focuses on restricting the capabilities of applications that a user believes may attempt malicious behavior. The Java sandbox is a well-known commercial example of a security model that is designed to restrict the behaviors of untrusted software. The Janus prototype, described in [9; 17], is one of the better-known research efforts that investigated performing application sandboxing. This and other systems of defending against malicious software are further described in the *Related Work* section.

In contrast, a *data-centric* approach to malicious software prevention focuses on better protecting resources from misuse by any application executing on the system. In the case of the FileMonster, the emphasis is on providing users with an extended set of file permissions that can be used to further safeguard important files. In a data-centric approach, the user chooses to protect important data from tampering or snooping. In an application-centric model the user must decide whether an application should be run in a restricted environment and how that application should access system resources. Our prototype is more limited in the type of resources that are protected (focusing only on the file system) however we believe that this results in a tool that is much easier to configure and use, and focuses on the resources that are most important to protect. We believe that a data-centric approach is not only easier to manage and understand, but that it is inherently safer than an application-centric model because protections is placed around the sensitive resource, not around the untrusted application. This paper will describe the data-centric malicious software defense prototype that we built and will discuss its various strengths and weaknesses.

3. Security vs. usability

As with almost any security mechanism there is generally a trade-off between security and usability. The FileMonster allows a user to vary a number of security settings that balance security with usability until an acceptable compromise has been reached. Here we explain FileMonster features that can be configured by a user and how these features affect the security and usability of the system.

3.1. Handling user confirmations

When a user marks a file such that confirmation is required, the FileMonster prototype can request this confirmation in one of two ways. The first confirmation method, which we will refer to as *simple file confirmation*, will simply block file access from occurring, then pop up a dialog box that asks the user to confirm or deny the action. As shown in Figure 2, this dialog box lists the file that is being accessed, the type of access (read or write) that is being requested, and the application that has issued the request. The user may either elect to allow the operation to continue as requested or to reject the request.

Using a simple dialog box to receive a confirmation from a user may be sufficient in most situations, however there is a security weakness in this approach that makes it unsuitable for some high security environments. The flaw is due to the ability of a hostile application to send windowing messages directly to the confirmation dialog box without any input from the user. If a piece of malicious code were designed specifically to thwart the FileMonster's protections, the code could attempt to access a protected file, wait for the confirmation window to pop up, and then send a confirmation message to that window that appeared to come from the user.

Figure 2. FileMonster confirmation window

Our solution to this problem is to provide users with the option of using a *secure file confirmation* method. This technique leverages Windows NT's built-in support for multiple desktops and the ability to secure these desktops. A desktop is "an on-screen work area that uses icons and menus to simulate the top of a desk" (Microsoft Visual Studio Help). An example of where Windows uses multiple desktops is the screen change that occurs when a user presses Ctrl-Alt-Del. Regardless of what you are working on, the screen is changed to reflect a new desktop with only a dialog box that presents options such as *Shutdown*, *Lock Workstation*, etc.

When an application creates a desktop it can control a variety of security settings for that desktop. When the FileMonster performs a *secure file confirmation* it creates a desktop that only the FileMonster program itself can manipulate. No other program is capable of sending messages to this desktop or using it to display windows. To perform a file confirmation, a user must switch to this desktop and then choose to allow or deny the requested operation. The FileMonster system can be sure that the confirmation comes directly from the user because no other programs can pass messages to the dialog box displayed on the secure desktop.

Providing both a *simple file confirmation* method and a *secure file confirmation* method allows users to choose their desired level of security. The *simple file confirmation* is vulnerable to attacks specifically targeting FileMonster's protection mechanism, however for a more secure environment a user can choose the *secure file confirmation* method. The *simple file confirmation* is slightly easier for someone to use because it does not necessitate switching desktops to perform a confirmation. Secure file confirmation ensures that even a piece of malicious code specifically designed to attack the FileMonster cannot bypass the security we have put in place.

3.2. Application file associations

Another feature that provides some trade-off between security and usability is the association of file types with applications. This feature enables the FileMonster to treat protected files differently depending on the application that is accessing them. To make an association, you first select either an individual file or file type and the application with which it is to be associated. Then you choose what permissions, if any, should be used when that application attempts to access the file or file type. At the time of the check the FileMonster uses MD5 hashing to verify that the application is indeed the program that was originally associated with the file or file type, and to see that this program has not since been modified.

For example, FileMonster could be configured such that all files with the *TXT* extension are treated with *confirm on write* permission regardless of which application is accessing it. You could use application associations to make an exception to this rule, stating that *TXT* files should not require any type of confirmation when access by the *Notepad* application. If any application other than *Notepad* attempted to write to a *TXT* file then a confirmation would be required, however when *Notepad* wrote to a *TXT* file the FileMonster would not interfere. Note that the method we are using to identify writing to a file will also identify an attempt to change the name or extension of that file. This prevents the simple attack of changing a file's extension before attacking its contents.

The security weakness that application associations introduce is in providing a path that bypasses FileMonster's own security permissions. In the example above, if the *Notepad* application contained some malicious logic that overwrote all of the *TXT* files on a hard drive there would be nothing in place to stop it.

Furthermore, a malicious program could take advantage of *Notepad's* file type association and perform its malicious actions through the *Notepad* application. Similarly, if an application that supports macros, such as Microsoft Word, were associated with a file or file type, then a malicious macro could control the host application and take advantage of the association to damage protected files.

Application associations are a double-edged sword. They help to reduce the number of confirmation dialogs that FileMonster generates, but could conceivably open up a hole in the prototype's armor. They must be used carefully, with the understanding that their convenience is paid for by the introduction of potential security weaknesses.

3.3. Session caching

The FileMonster allows users to enable a feature called *session caching* that will remember a user's response to a confirmation dialog box for as long as the application continues to run. If, for example, a user confirms that Microsoft Word is allowed to write to the file *MyFile.doc*, then until Microsoft Word is exited it will be allowed to write to that file without requiring another confirmation. This is very useful when someone is editing a protected document and will be saving the document frequently. If a different application tries to access the *MyFile.doc* file after a response has been cached it will still require the user to confirm the action.

It is possible that a malicious program might perform a malicious action at some point after the user has already cached a confirmation. This is a not a very significant threat because the user has already chosen to trust this application the first time. Additionally, there would be little to distinguish the first attempt to access a file as benign and a subsequent attempt as malicious.

3.4. Evaluating the FileMonster

For the FileMonster to be a useful tool it must provide protection against malicious software while maintaining a low profile. If the user is frequently asked to confirm actions then they will quickly begin to ignore FileMonster dialog boxes or to turn the tool off entirely. This has been demonstrated before with security features such as Microsoft Internet Explorer's warnings about accepting cookies. Though a potentially useful security feature, the frequency of the warnings causes most users to turn it off.

The goal of our work is to tune the FileMonster to the point where it produces a minimum number of false alarms, while still maintaining its effectiveness against malicious threats. We consider a false alarm to be a confirmation request that is caused by normal benign system use. To achieve this goal we instrumented the prototype with some basic logging capabilities. The FileMonster records its uptime and logs the date and time that every user confirmation is requested. A small set of users was selected as a test set and a standard configuration policy was used. These users were given basic training in the use of the FileMonster, and were taught to distinguish between common false alarms and likely malicious threats. The application associations and file type permissions used during this evaluation are shown in Table 1.

Table 1. FileMonster evalutation configuration

File Extension	Permissions	Application Associations (*Ignore on Access*)
.DOC	Confirm on Write	Microsoft Word Microsoft Outlook
.PPT	Confirm on Write	Microsoft PowerPoint
.EXE	Confirm on Write	(Microsoft Visual Studio Linking utilities)
.XLS	Confirm on Access	Microsoft Excel
.SKR (PGP Secret Key Ring Files)	Confirm on Access	(PGP Utilities)

This policy protects many of the basic file types that a Microsoft Windows user encounters on a daily basis. It assumes that the Microsoft applications listed are trusted, and that they are not being manipulated by malicious software. Throughout the test we employed the usability features described above (such as session caching). During this test phase users were asked not to adjust the configuration from its initial setting.

We logged FileMonster activity across our test set of users for approximately two months. Test subjects continued to use their computers for normal day-to-day activities. During this time period we found that the FileMonster resulted in an average of 1½ dialogs during a 24-hour period. We assume all of these confirmation requests to be false alarms because neither our corporate antivirus solutions nor the test users identified a malicious attack during this time frame (it would be ideal to test for false alarms in sterile environment, but the duration of these tests made this difficult).

It is difficult to quantify an acceptable number of spurious confirmation requests, but the results we were getting (about 1½ alerts per day) seemed too numerous for the FileMonster to be truly unobtrusive. Discussions with test candidates indicated that the FileMonster tended to seek user confirmation during web browsing sessions using Internet Explorer. Further examination narrowed

this down to sessions where Microsoft Word, Excel, and PowerPoint files were being viewed within the Internet Explorer browser. The dialog boxes were triggered when a new IE instance attempted to overwrite or delete temporary files created by a previous instance.

A quick fix was made to the FileMonster to ignore Internet Explorer's temporary files. This could be done easily because these files are always stored in the same location. A better solution to this problem is extending the FileMonster to allow users to specify rules such as these on-the-fly. After making these changes we resumed our evaluation. This simple change had a dramatic effect on the number of false alarms. During a test period of similar length we found that the number of dialogs was reduced to an average of one per week. This is a significant improvement over the original results and we believe that we can continue to push this number even lower by determining the cause of other false alarms.

4. How FileMonster works

To implement FileMonster's file access confirmation feature we need a way of detecting when a process is about to read from or write to a file. One way this could be done is to examine each function call that is made to *WriteFile* and *ReadFile*. Although this would give us very fine-grained control over a process's file manipulation, it would result in our having to intercept a very large number of function calls (these are two extremely heavily used functions).

An alternative to intercepting the individual attempts to read and write to a file is to regulate the type of access that is permitted when a process gets a handle to a file. On the Windows NT/2000 operating system all file access occurs through kernel file system objects that are manipulated from user-space through file handles. When a user-level application requests a file handle it must specify at that time whether it wants permission to read from or write to that file.

A user-level application has access to many functions in the Win32 API that will return a file handle or result in data being written to a file. For example the user-level functions *CreateFile*, *OpenFile*, and *_open* all return file handles. Identifying all of the user-level functions that can access files could be rather difficult, but fortunately it is completely unnecessary. All file handles correspond to file system objects within the kernel, and access to these objects is controlled by the Windows NT/2000 kernel system call *ZwCreateFile*. Any user-level applications that want to manipulate files are transparently routed through this function within the kernel. Note that despite its misleading name, this function is not simply for creating new files. The *ZwCreateFile* function is used whenever a process needs to get a handle to a file for future read or write operations. The *ZwCreateFile* system call will be invoked prior to any type of file I/O.

One of the parameters that must be passed to *ZwCreateFile* indicates the type of access that the process is requesting. Valid access types include read, write, and query. By looking at the *ZwCreateFile* file function call we can determine what file an application is about to use and how it intends to use it.

4.1. System call interception

Having determined the function that will allow us to implement the FileMonster's file access confirmation feature we will now discuss how to go about intercepting this function call. For our prototype to be successful in protecting against malicious code it must be non-bypassable. This means that there cannot be any way for malicious code to circumvent or remove our function interception mechanism. We are also interested in intercepting file access from all processes running on the system, not just from select applications. All of these requirements indicate that the correct location for us to place our interception mechanism is within the Windows NT/2000 kernel.

The Windows NT/2000 kernel can only be modified through the installation of device drivers. Device driver installation is tightly regulated by the operating system and is restricted to administrative users. This ensures that as long as an administrator is not executing the malicious code, it will be unable to interfere with any kernel modifications that we make. Additionally the use of a device driver provides us access to internal operating system functions and data structures not accessible from user mode.

In Windows NT, user applications invoke system services by executing an interrupt instruction. Code in the kernel takes control of the machine in response to the interrupt and performs some activity for the calling process before relinquishing control. A kernel entity known as the dispatcher initially responds to the interrupt request, determines the nature of the interrupt, and calls a function to handle the request. Two tables in kernel memory describe the locations and parameter requirements of all functions available to the dispatcher. One table specifies handlers for user requests; the other specifies handlers for requests originating within the kernel. The calling process places information about the requested system service on the stack along with any parameters required for completing the operation.

Our method of controlling file manipulation relies on our ability to instruct the dispatcher to call a function that we have written when a user process invokes certain system services. This approach requires constructing a device driver that is loaded into the kernel either dynamically or as part of the boot sequence. When our

driver is loaded it modifies an entry in the table that the dispatcher relies on for handling interrupt instructions. In our case, we are interested in intercepting calls to the *ZwCreateFile* function. The modification of the dispatcher's table results in a call to our function instead of the intended call to *ZwCreateFile*. Our function will be called whenever a user-mode application tries to get a handle to a file. The signature of our function is identical to that of *ZwCreateFile*, so the kernel interface exported to applications is not altered.

Once the dispatcher calls our function the FileMonster determines whether or not user confirmation will be required. If confirmation is not required, or if confirmation is granted, then we invoke the original *ZwCreateFile* with the same parameters as the calling process. If the user elects to deny the request, then we return a value indicating that the function call has failed and set our flags to indicate that access has been denied. The application will not be able to differentiate between a function failure produced by the FileMonster and a normally occurring error. It will handle this error the same way that it would handle an attempt to access a file by a user that does not have permission to do so. In some cases this results in an application that attempted to open a file with read/write permission to default to trying to open it as a read only file.

4.2. Configuration

FileMonster's configuration settings are protected through the use of secure desktops as discussed in section 3.1. This is necessary to prevent malicious code from changing the configuration itself. If the configuration program were not run on a secure desktop, then malicious code could send messages to the configuration program tricking it into making unwanted policy changes. The settings are passed from the user-level configuration program to our kernel-level device driver through our device driver's interface. We can leverage Windows NT's own security mechanisms to ensure that only a process running with administrative privileges is allowed to pass information to our device driver. The FileMonster configuration program is implemented as a Windows NT service that runs with administrative privileges. Figure 3 is a screen shot of one of the FileMonster configuration windows.

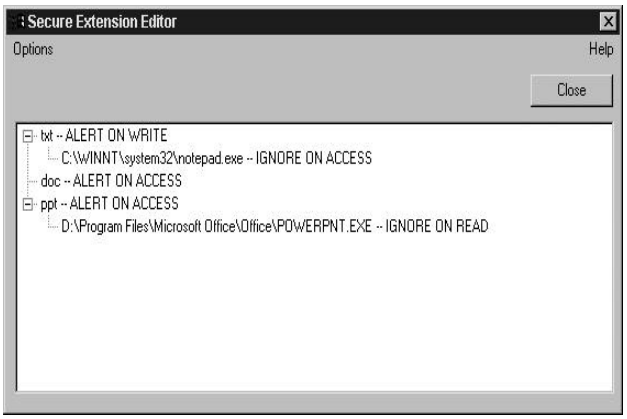

Figure 3. Configuring the FileMonster

To aid users in configuring the FileMonster we provide a mechanism for assigning default policies. The current incarnation of this tool enables users to choose one of three initial policies: *secure policy*, *basic policy*, or *no policy*. The *secure policy* establishes *confirm on read* and *confirm on write* policies for some of the most common Windows file formats. The *basic policy* extends the *secure policy* to include associations for commonly trusted applications. These applications are located using information stored in the registry and the policy is automatically built for the user. The final option is to begin with no policy and to build one from scratch. In all cases the policies can be fully modified from their default settings. Keep in mind that we are only concerned with protecting files that will be storing important user data. This is a small subset of the file types actually used in the Windows environment. To get an idea of how many file types are included in a typical policy, think of how many applications you use to create data that you would like to protect against damage or disclosure. For the users in our test environment we found this number to be below a dozen file types.

5. Related work

During the last decade there has been a lot of interest in the implementation and application of system call and function wrapping technologies. Some of this research has focused on providing flexible frameworks that facilitate the construction of systems such as the FileMonster. Other research has produced prototype systems that address the problems of controlling malicious software within a Discretionary Access Control environment. This section will examine this body of research and identify where the FileMonster fits into this collection of related work.

5.1. Wrapping techniques

Intercepting function calls or system calls is commonly known as *wrapping*. The basic idea of a wrapper is to provide functionality that will be called in place of the original target. This wrapper can perform any function including calling the original target function.

There are many potential uses for wrappers including extending the capabilities of an application, providing facilities for auditing, and restricting resource usage. One of the key difficulties in wrapping is developing a system that will work with applications without requiring the recompiling or relinking of this software. In [2] and [10] the authors present user-level wrapping techniques for Microsoft Windows operating systems. Hunt et al. describe a library that facilitates the wrapping of Windows API calls through the injection of *trampoline* code that redirects function calls at run-time. Balzer describes a similar system that has been additionally hardened against potential attackers. The goal of Balzer's work was to develop *non-bypassable* wrappers. This means that a malicious program cannot remove or circumvent a wrapper even if it is aware that it is present. Correctly implementing a user-level wrapping facility that cannot be bypassed by malicious code is extremely complex due to the myriad of ways that a file can be accessed (these are all reduced to the ZwCreateFile system call at the kernel level). We chose to implement our system at the kernel level because it provides us with the greatest degree of control over the file system.

In [15, 8, and 6] the authors build non-bypassable wrapper systems by intercepting system calls from within the Linux, Solaris, and FreeBSD/Solaris operating system kernels respectively. Like our approach, these techniques rely on the operating system's own security to protect the wrappers from tampering by user-level processes. Mitchem et al. discuss the usefulness of their system for secure auditing or to provide a fine-grained access control mechanism, but at this time they have not concentrated on the building of either of these systems. Unfortunately our decision to target the Windows NT/2000 platform eliminated the possibility of reusing these prototypes directly, however the work that they performed was helpful in designing our own interception mechanism.

5.2. Application Sandboxes

An application sandbox is an environment that restricts a process's resource usage. The resources that might be limited include the file system, network access, and even CPU and memory utilization. Sandboxing is a powerful technique for confining untrusted and potentially malicious software. Sandboxing systems are typically built around some sort of wrapping technology that gives them the fine-grained level of control necessary for them to be effective.

One of the most common examples of a sandbox is in the security built into the Java virtual machine. In [14] the authors discuss Java's sandbox security model and describe various attacks against it. Sandboxes are particularly useful for containing programs that have a high likelihood of containing malicious code because they can be used to severely restrict an application's capabilities. Mobile code is often considered to be untrustworthy, and is therefore an excellent candidate for sandboxing.

Unlike Java, most operating systems do not natively support application sandboxing. A number of research projects have investigated the use of sandboxes for restricting applications on UNIX and Windows operating systems. In [17] and [9], Wagner et al. introduce the Janus prototype that can be used to sandbox applications on the Linux operating system. While Janus focuses on restricting access to file system and network resources, in [5] the authors concentrate on limiting access to memory and CPU resources.

The work performed by Berman et al in [1] bears some similarity to our own. In this paper the authors present a process-specific file protection mechanism that they have implemented for the UNIX operating system. Their motivation for the development of this system is very similar to our own, however like most other sandbox efforts they focus their attention on applications, not on the data that is to be protected. In each of the sandboxing approaches that we have described, untrusted applications must explicitly be executed within a protection environment and file/directory permissions must be specified at the time of execution. In our opinion the extra effort required to run an application within these environment makes it unlikely that a user would consistently choose to do so.

Though it was never developed, in [13] Karger describes a system for controlling potentially malicious software in an operating system that supports discretionary access control. His proposal was to build a system that would use a file name translation mechanism to identify and prevent anomalous resource access. Similar to our approach, Karger recommended involving the user in arbitrating security decisions that the system itself could not make. As it was written, the proposed system was more appropriate for command-line driven operating systems, as of course was appropriate for this date of publication.

Macintosh users may be familiar with the GateKeeper utility written by Chris Johnson and described at [11]. This tool was intended to be a generic virus detection/protection mechanism that worked by monitoring an application's access to system resources. Whenever it detected an access that was considered

suspect it would query the user for confirmation before continuing. The approach that this tool takes is very similar to our own, however the resources that it monitors are rather different. The GateKeeper tool was concerned mainly with the protection of system files from viruses, not with the defense of user's documents. In a similar manner as the FileMonster, this utility functioned more as an additional access control mechanism for certain resources than as an application sandbox.

5.3. Where does the FileMonster fit it?

As we have discussed, a great deal of research has gone into the development of sandboxes and the underlying wrapping technology. It has been shown that wrapping can be performed on many different operating systems, and that it can be done in a secure manner. There is little question as to the potential of sandboxing as a defensive mechanism, however to this day it remains an under-utilized technology. The authors of this paper believe that the reason that sandboxing is not more popular is because although it is effective, the sandboxing mechanisms we described are often very difficult to configure and use. The FileMonster has been designed with these problems in mind.

We are not the first to recognize the need for more usable sandboxing technology. Recent work described in [4] presents a tool called WindowBox that provides a simplified sandboxing mechanism. This tool provides a form of sandbox separation between applications that run on different virtual desktops. The only way that information can be transferred from one desktop to another is with the explicit approval of the user. Belfanz et al. believe that this model provides users with an intuitive way to separate their applications and to protect them from each other. A potential problem with this approach is that all applications on a single desktop have full access to any data associated with that desktop. If a user is tricked into running a malicious program, this program will be able to damage whatever data it can access. To reap the benefits of this model it requires that users change how they go about their work, as well as necessitates that they concern themselves with the issue of how to group of applications on desktops. Though we take a different approach to the problem, this paper does represent a good effort to provide users with an easier to use sandbox environment and unfortunately we have not had the benefit of exploring it first-hand.

Though it shares some characteristics with application sandboxes, the FileMonster differs from most of the approaches described above in its focus on defending a user's documents rather than encapsulating untrusted applications. In this sense the FileMonster is closer to an extension of an operating system's access control mechanisms. We believe that our prototype protects file resources in a manner that is intuitive to the user and is significantly easier to manage. The data-centric model of protection allows a user to associate *confirm on read* and *confirm on write* permissions directly with the file or file type that is to be protected, rather than to have to decide which applications are dangerous enough to be sandboxed. By default all applications are subject to the restrictions set up by the FileMonster, making this an ideal system for protecting against malicious software that a user may not even realize is executing. Furthermore, the ability to require a *secure file confirmation* gives the FileMonster a security advantage over any application sandboxes that we have seen.

6. Discussion

The prototype described in this paper provides a unique solution to the established problem of controlling malicious software within a discretionary access control environment. Other research efforts in the areas of function wrapping and application sandboxing have provided the building blocks necessary to implement a solution to this problem, but have not produced a system that is both secure and easy to use. We hope that the FileMonster can help to fill this gap, and provide a much-needed layer of protection against damage or snooping by malicious software of all types.

The FileMonster provides *confirm on read* and *confirm on write* permissions to increase a user's awareness of an application's access to critical documents. The scope of our prototype's protection is more limited than that implemented by many application-centric protection measures that attempt to protect all types of resources. The FileMonster does not attempt to protect against nuisance attacks like denial of service attacks or email floods. Rather than being a disadvantage, we believe that this is critical to the success of the FileMonster. We have concentrated our protective measures on that which we believe is most important to defend. The FileMonster is most effective and least obtrusive when restricted to protecting important user documents. To this end we have provided a number of features including *session caching* and *application associations* that make it simple for a user to configure the tool to provide an appropriate level of additional security without interfering with normal work habits.

Because this prototype relies on the user to make security decisions the user must have a certain degree of security awareness. He must be able to distinguish between an ordinary file operation and a potentially malicious file operation. Usually the context of the operation provides enough data to make a sensible decision. For example, when a user elects to save the document titled *MyDocument.doc*, he should expect the FileMonster to present a *confirm on write* dialog box that

indicates that Microsoft Word is attempting to *write* to the file *MyDocument.doc*. In our experience using the FileMonster, we have found that most users do not have trouble making the requisite decisions. There will undoubtedly be situations that are not as straightforward, and the burden of making the correct decision will unfortunately fall on the shoulder of the user. We have yet to be able to explore this further in a larger test environment, but hope to make the FileMonster available for broader use in the near future and to leverage this experience to improve upon the current concept.

7. Future work

We will continue to improve the FileMonster by reducing the number of unnecessary confirmation requests and improving the user interface. The benchmarking that is discussed in section 3.4 will help us to quantify the improvements that we make to the system, but ultimately the success or failure of this prototype will depend on its ease of use and its unobtrusiveness. The usability studies that we have conducted so far have focused on reducing the number of spurious confirmation requests. This is an important element of usability because frequent dialog boxes will result in the user turning the FileMonster off, or not paying enough attention to catch actual attacks. Reducing the number of confirmations that a user has to respond to will increase the relative importance of each one. We hope to further evaluate usability in the near future by releasing a version of this tool that can be explored by the public at large.

One possible improvement to the security and usability of the tool is to introduce the use of hardware as a method of accepting user confirmations. A device driver could be written that would distinguish the difference between the user pressing a key on the keyboard and an application sending a "keystroke" to another application. This would enable the FileMonster to accept a confirmation request on the insecure user desktop without the need to switch to the secure FileMonster desktop. This would provide the security of using the *secure file confirmation* mode with the convenience of the *simple file confirmation* mode.

A known weakness in the protection that the FileMonster offers is the possibility of a malicious attack that manipulates a trusted application to read or damage protected files (this is discussed in section 3.2). We have not yet addressed this difficult problem, other than to caution against the use of application associations in high-security environments. One possible solution is to trap Windows system calls that relate to the passing of messages between applications. This would enable us to restrict the messages that are being sent to trusted applications. We could use this capability to prevent malicious software from manipulating trusted programs and attacking protected files.

The next property that needs to be evaluated is whether a user can easily differentiate between a benign confirmation request and one caused by malicious software. The FileMonster might be improved to actually help a user evaluate the seriousness of a confirmation dialog box. As an example of how this could be done, a high importance could be given to confirmation requests that originate from applications that are not part of a set of trusted applications. The easiest method of establishing this trusted code base would be to simply include all executables that were on the system at the time the FileMonster was installed, or to include all programs that have been configured as *application associations*.

Testing a user's ability to differentiate between malicious requests and benign requests necessitates a fairly involved experiment because we need to ascertain user's reactions to malicious software when they are not expecting to be attacked. Our experiments to this point have not included any actual malicious software attacks. In the future we hope to conduct a more comprehensive experiment; perhaps through a large-scale evaluation within the computer security community.

8. References

[1] A. Berman, V. Bourassa, E. Selberg, *"TRON: Process-specific file protection for the UNIX operating system."* In Proceedings of the 1995 USENIX Winter Technical Conference, pages 165--175. USENIX Association, 1995.

[2] R. Balzer, N. Goldman, "Mediating Connectors: A non-bypassable process wrapping technology." In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, pp. 361-368, 1999.

[3] M. Blaze, "A Cryptographic File System for Unix." In Proceedings of 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 1993, pp. 9--16.

[4] D. Balfanz, D. Simon, "WindowBox: A Simple Security Model for the Connected Desktop." In *Proceedings of the 2000 Windows System Symposium*, August, 2000

[5] F. Chang, A. Itzkovitz, V. Karamcheti, "User-level resource-constrained sandboxing." In *Proceedings of 4th USENIX Windows Systems Symposium*, 2000.

[6] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers." In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[7] C. Friberg, A. Held, "Support for Discretionary Role Based Access Control in ACL-oriented Operating Systems," In *Proceedings of 2nd ACM Workshop on Role Based Access Control*, pages 83--94. ACM, Fairfax, VA, November 6-7 1997.

[8] D. Ghormley, D. Petrou, Anderson, T. "SLIC: An Extensibility System for Commodity Operating Systems". In *USENIX 1998 Annual Technical Conference*, June 1998.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications --- Confining the Wily Hacker." In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[10] G. Hunt, D. Brubacher, "Detours: Binary interception of Win32 Functions." In *Proceedings of the 3rd USENIX Windows NT Symposium*, July, 1999.

[11] C. Johnson, "GateKeeper Version 1.3 Documentation." From http://gargravarr.cc.utexas.edu/gatekeeper/gatekeeper.html (May 29, 2001).

[12] T. Jeaeger, A. Prakash, and A. Rubin, "Building systems that flexibly control downloaded executable context." In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[13] P. Karger, "Limiting the Damage Potential of Discretionary Trojan Horses." In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 27-29, April 1987.

[14] G. McGraw, E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, 1999.

[15] T. Mitchem, R. Lu, R. O'Brien, "Using Kernel Hypervisors to Secure Applications." In *Proceedings of the 1997 Annual Computer Security Application Conference*, December 1997.

[16] C. Pfleeger. *Security in Computing*,. Prentice Hall, NJ, 1997, p. 270.

[17] D. Wagner. "Janus: an approach for confinement of untrusted applications." *Master's Thesis*, UC Berkeley, Computer Science Division, 1996.

* This article reflects the views of the authors only and does not necessarily reflect the views of the Department of Defense or the Defense Advanced Research Projects Agency.

Approved for Public Release, Distribution Unlimited